SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures

Yongjun Park, Sangwon Seo [*], Hyunchul Park [†], Hyoun Kyu Cho, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{yjunpark, swseo, parkhc, netforce, mahlke}@umich.edu

Abstract

Single-instruction multiple-data (SIMD) accelerators provide an energy-efficient platform to scale the performance of mobile systems while still retaining post-programmability. The central challenge is translating the parallel resources of the SIMD hardware into real application performance. In scientific applications, automatic vectorization techniques have proven quite effective at extracting large levels of data-level parallelism (DLP). However, vectorization is often much less effective for media applications due to low trip count loops, complex control flow, and non-uniform execution behavior. As a result, SIMD lanes remain idle due to insufficient DLP. To attack this problem, this paper proposes a new vectorization pass called SIMD Defragmenter *to uncover hidden DLP that lurks below the surface in the form of instruction-level parallelism (ILP). The difficulty is managing the data packing/unpacking overhead that can easily exceed the benefits gained through SIMD execution. The SIMD degragmenter overcomes this problem by identifying groups of compatible instructions (subgraphs) that can be executed in parallel across the SIMD lanes. By SIMDizing in bulk at the subgraph level, packing/unpacking overhead is minimized. On a 16-lane SIMD processor, experimental results show that SIMD defragmentation achieves a mean 1.6x speedup over traditional loop vectorization and a 31% gain over prior research approaches for converting ILP to DLP.*

Categories and Subject Descriptors D.3.4 [*Processors*]: [Code Generation and Compilers]; C.1.2 [*Processors Architectures*]: Multiple Data Stream Architectures—Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms Algorithms, Experimentation, Performance

Keywords Compiler, SIMD Architecture, Optimization

1. Introduction

The number of worldwide mobile phones in use exceeded five billion in 2010 and is expected to continue to grow. The computing platforms that go into these and other mobile devices must provide ever increasing performance capabilities while maintaining low energy consumption in order to support advanced multimedia and signal processing applications. Application-specific integrated circuits (ASICs) were the most common solutions for the heavy lifting, performing the most compute intensive kernels in a high performance but energy-efficient manner. However, new demands push designers toward a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

[*] Currently with Qualcomm Incorporated, San Diego, CA

[†] Currently with Programming Systems Lab, Intel Labs, Santa Clara, CA

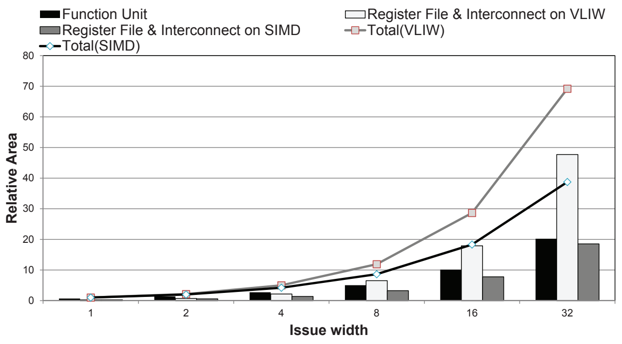


Figure 1. Scalability of datapaths that exploit instruction-level parallelism (VLIW) and data-level parallelism (SIMD). Plotted is the relative area as issue width increases from 1 to 32. Area is broken down into function unit and register file & interconnect.

Processors that exploit instruction-level parallelism (ILP) provide the highest degree of computing flexibility. Modern smart phones employ a one GHz dual-issue superscalar ARM as an application processor. Higher performance digital signal processors are also available such as the 8-issue TIC6x. However, the scalability of ILP processors is inherently limited by register file (RF) and interconnect complexity as shown in Figure 1. Single-instruction multiple-data (SIMD) accelerators have long been used in the desktop space for high performance multimedia and graphics functionality. But, their combination of scalable performance, energy efficiency, and programmability make them ideal for mobile systems as well [4, 5, 12, 20, 33]. Figure 1 shows that the area of SIMD datapaths scale almost linearly with issue width. Power follows a similar trend [33]. SIMD architectures provide high efficiency because of their regular structure, ability to scale lanes, and low control overhead.

The difficult challenge with SIMD is programming. The application developer or compiler must find and extract sufficient data-level parallelism (DLP) to efficiently make use of the parallel hardware. Automatic loop vectorization is a popular approach and is available in a variety of commercial compilers including offerings from Intel, IBM, and PGI. Applications that resemble classic scientific computing (regular structure, large trip count loops, and few data dependences) perform well on most SIMD architectures.

However, mobile applications are not limited to these types of applications. High-definition video, audio, 3D graphics, and other

Vectorization granularity

Coarser ◄·········► Finer

Level	Loop	Subgraph	Superword
Scope	Loop body	Group of instructions	Instruction
Vectorization advantage	High	Middle	Small
Coverage	Small	High	High

Figure 2. A spectrum of the vectorization at different granularities.

forms of media processing are high value applications for mobile devices. These applications continue to grow in complexity and resemble scientific applications less and less. Computation is no longer dominated by simple vectorizable loops. Instead, current media processing algorithms behave more like general-purpose programs with DLP available selectively and to varying degrees in different loops. Also, significant amounts of control flow are present to handle the complexity of media coding and limits the available DLP. The overall affect is that loop-level DLP is less prevalent and less efficient to exploit in media algorithms. Due to these application-specific complexities, available SIMD resources cannot be fully utilized and a substantial portion of resources are idle at runtime. Talla [32] reports that only 1-4% performance improvement exists when scaling the SIMD components from 2-way to 16-way on the MediaBench suite [19]. Thus, an improved approach beyond simple loop level techniques is necessary in order to effectively use wide SIMD resources.

To supplement the insufficient degree of DLP from traditional vectorization, superword-level parallelism (SLP) [17] can be applied. SLP is a short SIMD parallelism between isomorphic instructions within a basic block. As shown in Figure 2, SLP can cover more code regions as compared to loop-level vectorization because SLP can be performed in non-loop regions, in loops having cross-iteration dependences, and in outer loops. For vectorizable loops, traditional vectorization is preferred because SLP misses loop-specific optimization opportunities [24]. The weakness of SLP is that the vectorization scope is too fine, resulting in a high overhead of getting data into packed format that is suitable for SIMD execution. Often, this packing overhead can exceed the benefits of parallel execution on the SIMD hardware. In addition, SLP is performed with a local scope that commonly misses opportunities for vectorization when a large number of isomorphic instructions exist.

To address the limitations of SLP, we introduce a coarser level of vectorization within basic blocks, referred to as *Subgraph Level Parallelism (SGLP)*. SGLP refers to the parallelism between subgraphs (groups of instructions) having identical operators and dataflow inside a basic block: parallel subgraphs that can execute together on separate data. SGLP has two major advantages that allow it more opportunities to convert ILP to DLP: 1) data rearrangement and packing overhead can be minimized by encapsulating the data flow inside the subgraph, 2) natural functional symmetries that exist in media applications (e.g., a sliding window of data long which computation is performed) can be exposed to enable vectorization of larger groups of instructions. The net result is SGLP leads to a combination of more SIMD execution opportunities and fewer instructions dedicated to data reorganization and inter-lane data movement.

This paper presents the design of a supplemental vectorization pass referred to as the *SIMD Defragmenter*. It automatically identifies and extracts SGLP from vectorized loops and orchestrates parallel execution of subgraphs with minimum overhead using unused resources. In the SIMD Defragmenter, a loop is first vectorized using traditional vectorization techniques. Then, vectorizable subgraphs are identified based on the availability of unused lanes in the hardware. The compiler then allocates the subgraphs to unused SIMD resources to minimize inter-lane data movement. Finally, new SIMD operations for SGLP are emitted and operations for inter-lane movements are added where necessary. Small architectural features are provided to enhance the applicability of SGLP and the configuration is statically generated during compilation.

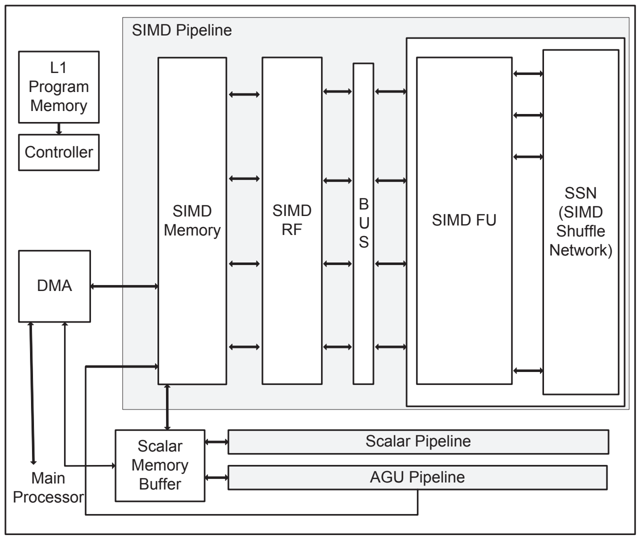


Figure 3. Baseline SIMD architecture.

This paper offers the following three contributions:

- An analysis of the difficulties of putting SIMD resources to efficient use across three mobile media applications (MPEG4 audio decoding, MPEG4 video decoding, and 3D graphics rendering).
- The introduction of SGLP that can efficiently exploit unused SIMD resources on already vectorized code.
- A compilation framework for SGLP that identifies isomorphic subgraphs and selects a mapping strategy to minimize data reorganization overhead.

2. Background and Motivation

In this section, we examine the current limitations of SIMD architectures based on an analysis of the following three widely used multimedia applications:

- AAC decoder: MPEG4 audio decoding, low complexity profile
- H.264 decoder: MPEG4 video decoding, baseline profile, qcif
- 3D: 3D graphics rendering

We then analyze why the well-known solutions are not as effective as expected. Finally, we discuss several potential approaches to overcome these bottlenecks and increase the utilization of existing resources.

2.1 Baseline Architecture Overview

A basic SIMD architecture that is based on SODA [20] (Figure 3) is used as the baseline architecture. This architecture has both SIMD and two scalar datapaths. The SIMD pipeline consists of a multiple-way datapath where each way has an arithmetic unit working in parallel. Each datapath has a two read-ports, one write-port, a 16 entry register file, and one ALU with a multiplier. The number of ways in the SIMD pipeline can vary depending on the characteristics of target applications. The SIMD Shuffle Network (SSN) is implemented to support intra-processor data movement. The scalar pipeline consists of one 16-bit datapath and supports the application's control code. The AGU pipeline handles DMA (Direct Memory Access) transfers and memory address calculations for both scalar and SIMD pipelines.

2.2 Analysis of Multimedia Applications

SIMD architectures provides an energy-efficient means of executing multimedia applications. However, it is difficult to determine the optimal number of SIMD lanes because the number depends on the algorithms that constitute the workload. In this analysis, we first categorize the innermost loops of three applications into different groups according to their vector width. Then, two types of SIMD width variance are identified and the practical difficulties of finding the optimal SIMD width and achieving high utilization are discussed.

2.2.1 SIMD Width Characterization

Multimedia applications typically have many compute intensive kernels that are in the form of nested loops. Among these kernels, we analyze the available DLP of the innermost loops and find the maximum natural vector width which is achievable. Based on the Intel Compiler [15], the rules to be selected as a vectorizable innermost loop are as follows:

- The loop must contain straight-line code. No jumps or branches, but predicated assignments, are allowed only when the performance degradation is negligible.
- The loop must be countable and there must be no data-dependent exit conditions.
- Backward loop-carried dependencies are not allowed.
- All memory transfers must have same strides over iteration.

If a loop satisfies the above four conditions, the minimum iteration count is set to the vector width of the loop.

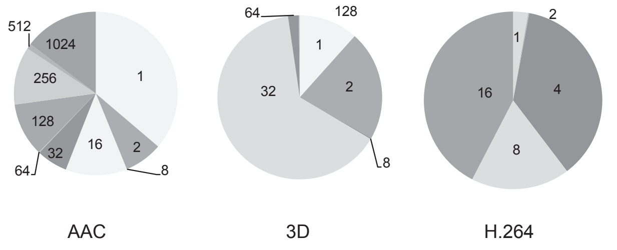


Figure 4. Scalar execution time distribution at different SIMD widths for three media applications: the maximum SIMD widths are 1024, 128, and 16, and the SIMD widths, which can be fully utilized for more than 50% execution time, are 16, 32, and 8 for AAC, 3D, and H.264 applications.

2.2.2 SIMD Width Variance

Figure 4 shows how many different natural vector widths reside in the three target benchmarks. The execution time breakdown between loops having different vector widths are shown in Figure 4. The three pie charts show the distribution of scalar execution time spent in innermost loops at various SIMD widths for three applications. From Figure 4, we can see that there are many different vector widths inside each application, hence it is quite difficult to determine the optimal SIMD width even for one application. For example, to define 16 as the SIMD width for H.264 is not desirable because the maximum vector width is 16 but the execution time ratio of loops with vector width of 16 is just 42% and some SIMD lanes are wasted for the remaining time. On the other hand, four is also not desired because the execution time of the loops with a width of four is not dominant with substantial execution occurring in loops having larger SIMD widths. Similarly, AAC and 3D applications cannot set the number of SIMD lanes as the maximum vector width due to the waste of resources, nor dominant vector width due to the low performance. Therefore, effectively supporting multiple SIMD widths is required to take advantage of the SIMD architectures.

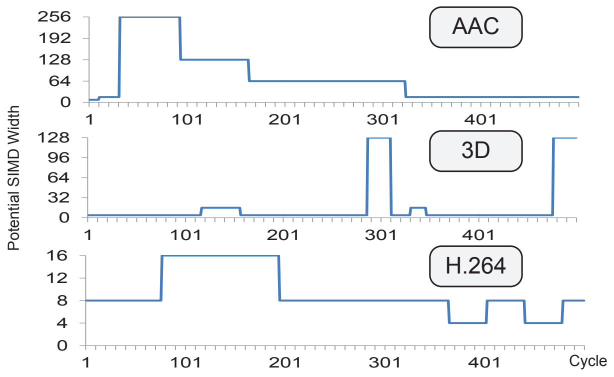


Figure 5. The SIMD width requirement changes at runtime: The X-axis indicates the execution clock cycle and the Y-axis is the maximum SIMD width assuming infinite resources. The minimum duration between width transition is 20 cycles from 311 to 330 for 3D application.

Dynamic power gating is one of the most successful energy saving techniques for the resource waste problem. Each SIMD lane can be selectively cut off from the power rails when the lane is not utilized using a MOSFET switch. This technique is attractive because it is effective for dynamic power saving and also has positive impact on leakage power savings. Although dynamic power gating achieves high energy savings, the relatively high overhead when changing modes prevents current SIMD architectures from applying it [29]. Even applying simple dynamic power gating techniques [14, 22, 23] is not effective since at least a few microseconds are required to compensate the power on/off energy overhead in current technologies. Figure 5 shows the SIMD width requirement changes over the runtime for three applications. The x-axis is the time stamp for 500 cycles when the SIMD architecture supports infinite DLP and the y-axis is the natural SIMD width that achieves the best performance. As shown, power gating cannot even compensate the transition energy overhead because of frequent power mode transitions within less than 200 cycles (1 μs at 200 Mhz) based on the different SIMD width requirements. Moreover, power gating comes with about 8% area overhead due to the header/footer power gate switch implementation. Therefore, power gating is hard to integrate into current SIMD architectures.

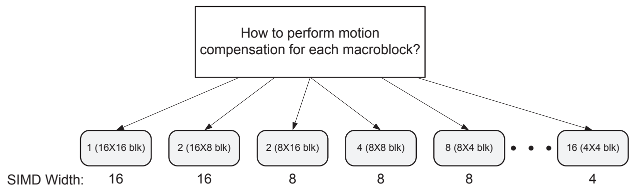
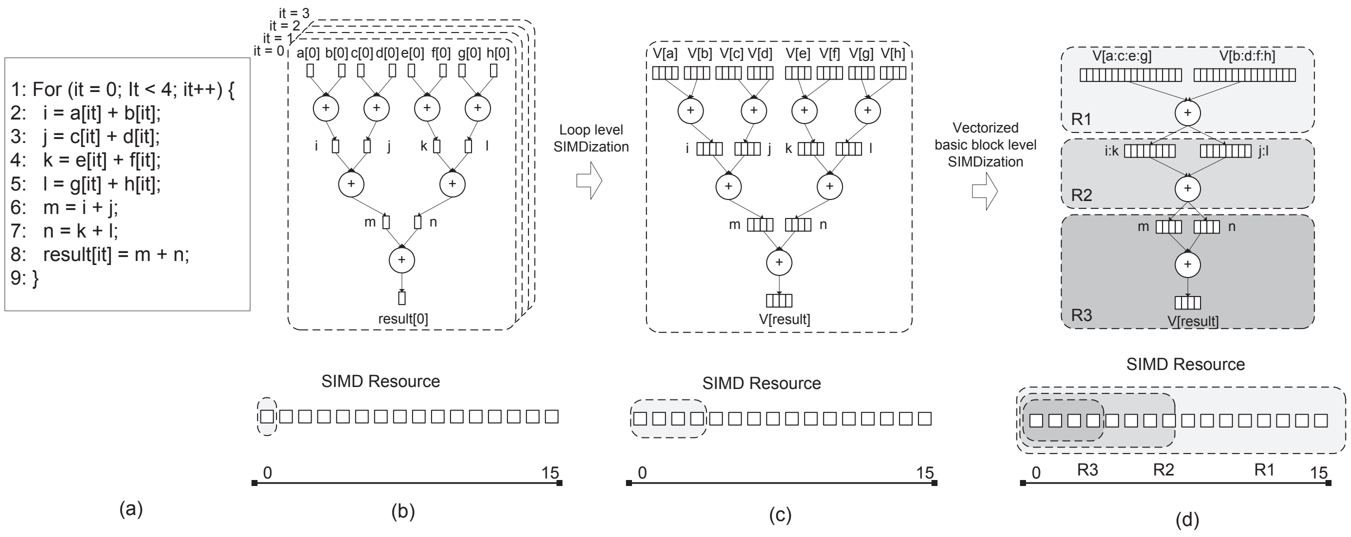


Figure 6. Different SIMD width requirements for each macroblock in the motion compensation process in H.264 decoder. The information is provided at runtime.



```
For (it = 0; It < 4; it++) {
  i = a[it] + b[it];
  j = c[it] + d[it];
  k = e[it] + f[it];
  l = g[it] + h[it];
  m = i + j;
  n = k + l;
  result[it] = m + n;
}
```

Figure 7. Different levels of parallelism: (a) an example loop's source code, (b) original multiple scalar subgraphs utilizing a single SIMD lane, (c) a vectorized subgraph using four SIMD lanes, and (d) the opportunity of partial SIMD parallelism inside the vectorized basic block (SIMD lane utilization: (R1: 16), (R2: 8), (R3: 4))

Thread-level Parallelism (TLP) for SIMD architectures has also been proposed to solve the temporal resource waste due to the small amount of DLP [34]. TLP supports running multiple threads that work on separate data on a wide SIMD machine when the SIMD width is small. By exploiting two kinds of parallelism, the SIMD lanes can be maximally utilized but the realization of TLP's potential in SIMD architectures has some critical limitations. First, TLP might not be fully exploited if parallel threads have different instruction flow. The motion compensation process for the H.264 decoder is a well-known example of this case. Figure 6 shows the various configurations of the motion compensation process for one macroblock. In this figure, the configuration of each macroblock is different so that SIMD specific restriction, which needs to execute the same instruction stream across the lanes, prohibits executing multiple processes in parallel even though the process has high TLP. Second, TLP cannot handle input-dependent control flow. For example, conditions to choose the macro block configuration in Figure 6 are decided from input header data hence TLP cannot be considered in the compilation phase. Finally, TLP generally requires more memory pressure. As a result, TLP looks appealing but the actual implementation of it is complicated.

The analysis reveals the difficulty of implementing common solutions in the real world. To further improve resource utilization, it is necessary to find a way to exploit other forms of parallelism.

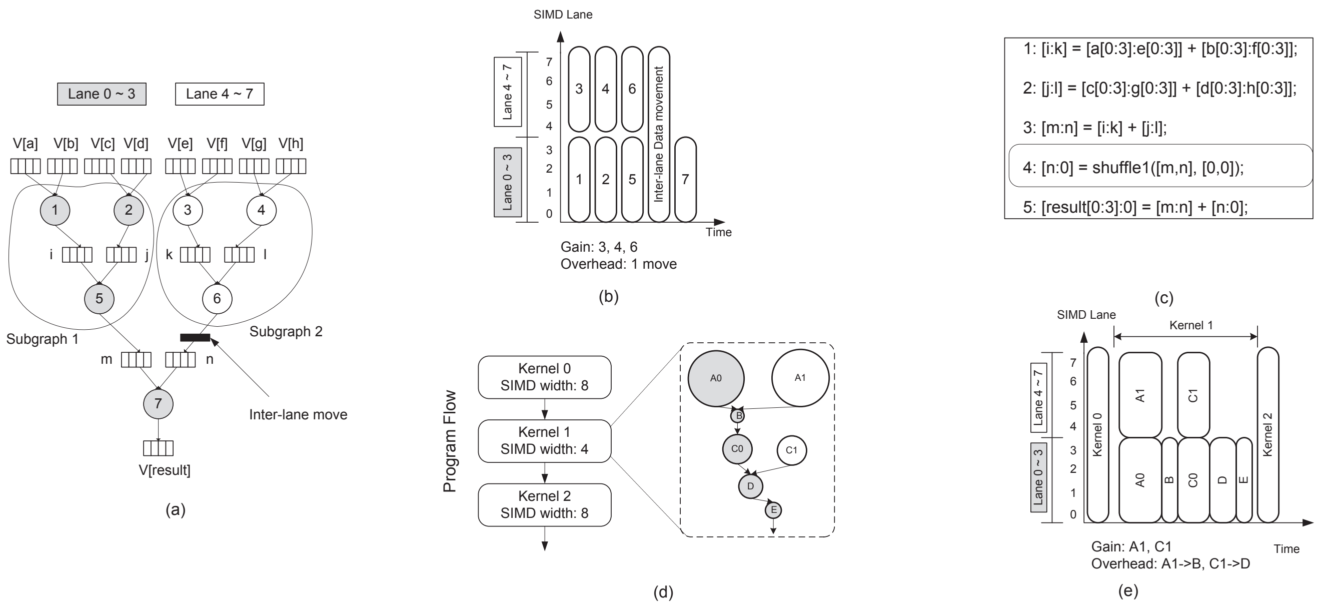
2.3 Beyond Loop-level SIMD Parallelism

Most kernels have some degree of DLP, which can be easily vectorized using loop unrolling. An interesting question is how to find extra parallelism when the degree of DLP is smaller than the degree supported in the architecture. For this question, the next opportunity can be found inside the vectorized basic block. Even if the basic block is not fully vectorizable, some parts inside the block can be vectorized as a restricted form of ILP. Compared to ILP, DLP requires two more conditions: 1) the instructions should perform the same work and 2) data flow should also be in the same form. Therefore, parallel instructions with the same opcode can be executed together in a SIMD architecture. Figure 7 shows examples of additional SIMD parallelism inside the vectorized basic block for our three applications. Figure 7 (a) is a vectorizable loop to generate the sum of eight input data arrays. (b) shows the unrolled dataflow graph (DFG) that can be executed in only one lane when assuming a 16-way SIMD datapath. This loop can then be vectorized as shown in (c) and four lanes can be assigned as the trip count of the loop but still 12 lanes are idle. In this case, another opportunity for partial SIMD parallelism can be found inside the basic block as illustrated in (d). Four ADD instructions in the 'R1' region are able to execute together with 16 degrees of parallelism, two ADD instructions in the 'R2' region can also execute together using eight lanes. Based on the application analysis, more than 50% of total instructions have at least one parallel identical instruction.

2.4 Summary and Insights

The analysis of these three applications provides several insights. First, resource utilization of a wide SIMD architecture is low because multimedia applications have various degrees of SIMD parallelism, and current solutions are not effective due to the high dynamic variance and the unpredictability. Second, ILP inside the vectorized basic block can be converted to DLP in many cases. Therefore, additional partial SIMD parallelism can be added when the DLP is insufficient.

A major challenge is how to minimize the data movement across the different SIMD lanes. For loop-level DLP, inter-lane data movement does not happen, whereas partial DLP has a large number of such movements due to each part having different levels of DLP, causing the amount of the occupied SIMD lanes to change at runtime in such a manner that the data packing/unpacking/reorganizing process happens frequently. For example, two data movements across the lane need to be done when exploiting partial SIMD parallelism in Figure 7 (d): 1) 'R1' to 'R2': After the 16-wide instruction, half of the data in lanes 9 to 15 should move to 0 to 8, and 2) 'R2' to 'R3': After the 8-wide instruction, half of the data in lanes 4 to 7 should move to 0 to 3. Therefore, we can save just two total instructions due to the data movement even if we save four instructions on partial SIMD parallelism. The conclusion is that minimizing inter-lane data movements is the key challenge in getting benefits from partial SIMD parallelism.



```
1: [i:k] = [a[0:3]:e[0:3]] + [b[0:3]:f[0:3]];

2: [j:l] = [c[0:3]:g[0:3]] + [d[0:3]:h[0:3]];

3: [m:n] = [i:k] + [j:l];

4: [n:0] = shuffle1([m,n], [0,0]);

5: [result[0:3]:0] = [m:n] + [n:0];
```

Figure 8. Subgraph level parallelism: (a) identical subgraphs are identified, and (1, 2, 5, 7) and (3, 4, 6) are executed in parallel with one overhead, (b) execution of the graph on two SIMD lane groups, (c) SGLP exploited output source code, (d) high level program flow with three sequential kernels and kernel 1 can exploit SGLP, and (e) execution of three kernels with SGLP exploration on kernel 1.

3. Subgraph Level Parallelism

This section describes a new vectorization technique. We first introduce some new terminologies and discuss its effectiveness in contrast to other related techniques. An execution model using SGLP is then proposed on the conventional wide SIMD architecture. Finally, we list practical challenges to exploit this parallelism and suggest proper solutions.

3.1 Overview

Subgraph level parallelism is defined as SIMD parallelism between *identical* subgraphs which 1) have an isomorphic form of dataflow with SIMDizable operations and 2) have no dependencies on each other inside the basic block. This parallelism is detected through the identical subgraph search inside the whole dataflow graph extracted from a basic block. These identical subgraphs can be executed in parallel in the form of a sequence of SIMD instructions inside the subgraph. There are two major advantages when searching packing opportunities at the subgraph level:

- **Packing steering:** SGLP minimizes the overall data reorganization overhead because the data movements between instructions inside a subgraph are automatically captured and assigned to one SIMD lane, and the alignment analysis between subgraphs is performed over a global scope. This benefit becomes more apparent when converting ILP to DLP in the low-DLP region such as loop-level vectorized or scalar code because the subgraph guides the instruction packing in directions that reduce or keep constant the amount of conversion overheads when the packing opportunities are not restricted by the memory alignment so that the number of possible packing combinations increase.
- **High packing gain:** Converting ILP to DLP is not common because it is hard to expect that the data reorganization process will provide enough gain to compensate for its performance loss due to the expensive nature. However, the considerable instruction savings of subgraph packing gives more chances to guarantee a positive net performance gain in spite of the substantial amount of overheads.

Figure 8 illustrates an example of SGLP realization. Using the vectorized basic block from Figure 7, Figure 8(a) identifies two identical subgraphs of (1, 2, 5) and (3, 4, 6) due to the same dataflow and same operations with no dependencies. Each corresponding instruction of two subgraphs is packed together and executed in parallel. Figure 8(b) shows the actual execution model using an 8-way SIMD datapath. Due to the insufficient degree of DLP for the original innermost loop from Figure 7(a), SGLP is applied and two isomorphic subgraphs are identified from the 4-wide vectorized basic block (Figure 8(a)). From these two subgraphs, (3, 4, 6) is chosen to be executed in the unused lanes. As a result, instructions (1, 2, 5, 7) and (3, 4, 6) are executed in lane 0-3 and 4-7 as shown in Figure 8(b). In addition to this, one cycle of overhead is incurred to move the output data of instruction 6 to lane 0-3. Figure 8(c) is the pseudo code exploiting both SIMD parallelism and SGLP. In this program, parallel instructions in the isomorphic subgraphs are packed together and data movement is enabled by the *shuffle* instruction, which moves data using the shuffle network in Figure 3. Shuffle0 extracts the left column data of two input vectors and Shuffle1 extracts the right column data of two input vectors.

Figure 8(d) and (e) illustrate the high-level execution model of this paradigm. The example scenario is three consecutive kernels having different natural SIMD widths (kernel 0:8, kernel 1:4, kernel 2:8) are executed on an 8-way SIMD architecture. Kernel 0 and 2 are executed only using SIMD parallelism by loop unrolling without any inter-lane overhead. However, the natural SIMD width of kernel 1 is smaller than the architecture allows, so SGLP is exploited as shown in (d). The SGLP compiler finds two groups of two isomorphic subgraphs as (A0, A1) and (C0, C1) and offloads two subgraphs of A1 and C1 onto lanes 4-7. As a result, the whole program can improve the total performance by the execution time of A1 and C1 as shown in (e) with some overhead. Inspired by this scenario, the total speedup achieved by SGLP over the current execution model is derived as the following equation when executing

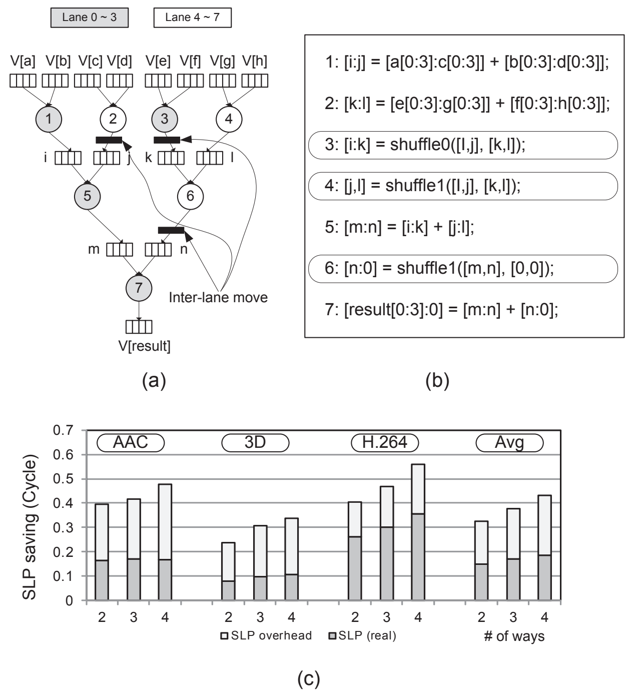
n different kernels with iv invocations, which have t normal execution time, t_{sglp} execution time can be saved by subgraph offloading and ov inter-lane movement overhead.

$$Speedup = \frac{\sum_{k=0}^{n-1}(t(k) \times iv(k))}{\sum_{k=0}^{n-1}((t(k) - t_{sglp}(k) + ov(k)) \times iv(k))} \quad (1)$$

Based on Equation (1), the performance gain can be maximized when a program has a high number of invocations on kernels with a small degree of DLP, a high degree of SGLP and a small overhead. Therefore, an SGLP compiler needs to increase the number of instructions covered by identical subgraphs with minimum inter-lane overhead. The key algorithm to achieve this goal is explained in section 4.

3.2 Comparison with Superword Level Parallelism

Superword level parallelism [17] is the most similar paradigm to our work with respect to searching potential parallelism inside the basic block. Because SLP focuses on short SIMD instructions, isomorphic instructions are only considered and thus they cannot handle inter-lane data movement. This problem is often ignored because the overhead of data movement inside the vector is fairly small in a narrow SIMD component, however, it usually induces high performance degradation in a wider SIMD component [17]. In addition to this, the local scope of superword level parallelism may be fooled into selecting packing instructions when a large number of isomorphic instructions exists.



```
1: [i:j] = [a[0:3]:c[0:3]] + [b[0:3]:d[0:3]];
2: [k:l] = [e[0:3]:g[0:3]] + [f[0:3]:h[0:3]];
3: [i:k] = shuffle0([l,j], [k,l]);
4: [j,l] = shuffle1([l,j], [k,l]);
5: [m:n] = [i:k] + [j:l];
6: [n:0] = shuffle1([m,n], [0,0]);
7: [result[0:3]:0] = [m:n] + [n:0];
```

Figure 9. Superword level parallelism difficulty: (a) (1, 3, 5, 7) and (2, 4, 6) are chosen to execute in parallel and three overheads occur, (b) superword level parallelism exploited output source code, and (c) average cycle savings of SLP: Y-bar means ideal savings and it is broken down as overheads and real savings.

Figure 9 shows the result of exploiting superword parallelism from Figure 7 (a). For a fair comparison, we relax the memory alignment constraint of [17] so that all memory instructions can be packed. As the compiler searches the isomorphic instructions in program order with local scope, instructions are packed as (1, 2), (3, 4), (5, 6). Then lanes 0-3 execute (1, 3, 5, 7) and lanes 4-7 execute (2, 4, 6) as shown in Figure 9 (a). Even though total instruction savings are the same as SGLP , the overhead also increases to three instructions (Figure 9 (b)). Therefore, there is no performance gain even in this small basic block, and when the block becomes more complex the algorithm cannot ensure a good result.

Based on the above consideration, we analyze the cost of these overheads for the vectorized kernels of three media applications. Figure 9(c) shows average cycle savings when applying SLP at different SIMD ways from two to four compared to the original schedule on the baseline processor. The Y-bar shows the ideal savings assuming the SIMD overhead is free, and each bar is broken down by SLP overhead and real savings. The SLP overhead is calculated assuming all the data rearrangement instructions take one cycle. The results give two major insights: 1) SLP, the well-known SIMDization technique used inside the basic block, can ideally deliver a fair amount of performance enhancement and is also scalable as the number of ways increases, and 2) large SIMD overheads of more than 50% of ideal savings hinder the effectiveness of SLP and make SLP barely scalable as the overheads also grow dramatically at wider ways. The actual performance gain will be worse in a real situation because many SIMD overhead instructions take more than an single cycle with current technology. Section 5 shows how much SGLP improves performance by both increasing the ideal savings and decreasing the overheads when compared to SLP. In addition to this, we also show how much ILP can be converted into SGLP.

3.3 Challenges and Solutions

As discussed, SGLP introduces more potential parallelism but has many principal challenges to make this paradigm feasible. We list the four major architectural challenges and suggest possible solutions with architectural or compiler modifications. Simple architectural changes are proposed as shown in Figure 10 and compilation challenges are addressed in Section 4.

Control flow: Because SGLP is basically exploited within the basic block, control flow is not a big issue. Furthermore, as scalar pipelines are primarily responsible for handling control flow, SGLP generally does not need to consider control flow. However, basic blocks are sometimes merged using if-conversion with predication. Even in this case, SGLP is not affected because predication also can be detected in the identical subgraph identification process.

Instruction flow: When multiple SIMD lane groups execute some task in parallel, all the instructions are not covered as subgraphs, and some SIMD lane groups may not be enabled because the number of identical subgraphs is smaller than the number of SIMD lane groups. Therefore, the main SIMD lane group is necessary to cover all the instructions.

Register flow: First, data movement across or inside the SIMD lane groups can be supported by single-cycle shuffle instructions using a shuffle network. Second, although multiple SIMD lane groups execute the same instructions, their actual register names are different. Therefore, the compiler must handle register renaming, which packs multiple parallel short registers into a wide register. In addition to this, some instructions covered by multiple identical subgraphs may have different immediate values, and therefore the architecture must provide a way to support wide constant values in a cycle because it is impossible to supply multiple values in a cycle. Therefore, a small constant value memory can be added. The compiler then automatically generates the wide constants from multiple immediate values. The application study shows that these cases rarely exist, and thus the overhead incurred is trivial.

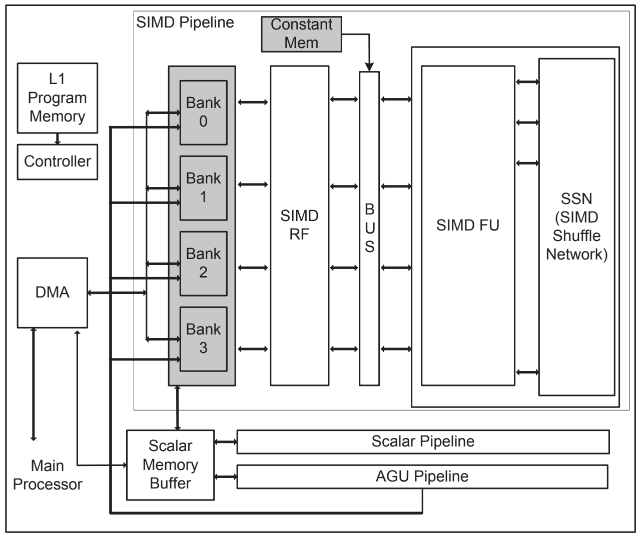


Figure 10. Architectural modifications: (1) multi-bank memory and (2) wide SIMD constant memory is supported.

Memory flow: If identical subgraphs have memory instructions, the references of the instructions may be different, and thus the architecture must provide a smart memory packing mechanism such as gather-scatter operation.

The possible architectural modification is to replace the SIMD scratchpad memory from one wide memory to a short width multiple bank memory. This change is required to relax the memory alignment constraint. The most critical reason why the basic block typically has high ILP but low DLP is that the architecture does not support unaligned memory access [17]. By supporting unaligned memory packing/unpacking from DMA using the multi-bank memory, more memory instructions can be executed in parallel. One key point is that multi-addressing is only allowed for Memory-DMA communications, while the SIMD pipeline views the memory as a single bank. Another key point is that the number of banks depends on the ratio of the number of memory instructions to normal instructions because the address calculations are the responsibility of the AGU pipeline and they are not scalable, thus the performance of the AGU may be the limiting factor.

4. Compiler Support

4.1 Overview

In this section, we describe the compiler support for SGLP. Taking the concept of subgraph identification [11], we developed a SGLP scheduler that can support both simple loop-level DLP and SGLP for wide SIMD components. The system flow is shown in Figure 11. Applications are run through a front-end compiler, producing generic Intermediate Representation (IR), which is unscheduled and uses virtual registers. The compiler also gets high-level machine specific information, including the number of SIMD lanes, and supported inter-lane movement instructions. Given the IR and hardware information, the compiler performs loop-level vectorization on the selected SIMDizable loops. The compiler then exploits SGLP if the SIMD parallelism is insufficient. After generating the DFG, the compiler iteratively discovers identical subgraphs in the DFG and assigns the subgraphs to unused SIMD lanes until no more SGLP opportunities exist. Finally, the compiler generates the final vectorized IR.

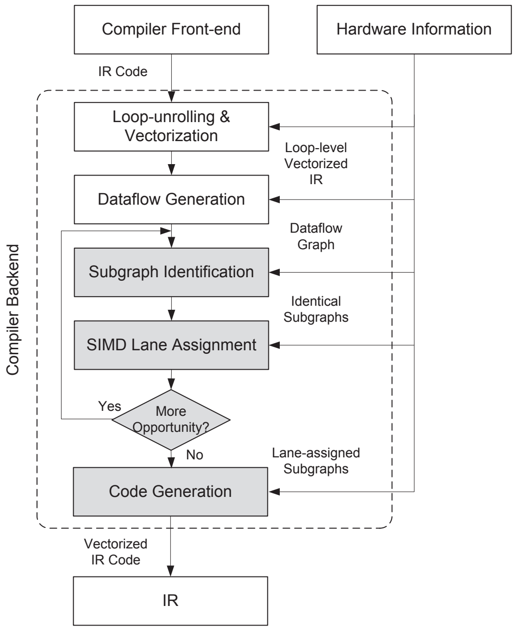


Figure 11. Compilation flow of the SIMD defragmenter: shaded regions exploit subgraph level parallelism.

4.2 Subgraph Identification

First, identical subgraphs are extracted from the given DFG. The compiler sets the maximum number of identical subgraphs as the available degree of SGLP. The compiler then iteratively searches the groups of identical subgraphs having some number of instances from maximum number down to two (the minimum degree). Heuristic discovery [11], which picks the seed node and grows the nodes, is used for DFG exploration. Exploration starts by examining each node in the DFG and using it as the seed for a candidate identical subgraph. The algorithm attempts to find the largest candidate subgraphs with n identical instances within the given DFG, where n is the degree of SGLP. If, however, the algorithm identifies m identical instances of a candidate subgraph, where $m > n$, only n instances are saved and the nodes from the remaining $m - n$ instances are “discarded” and “re-used” in the next exploration phase. This of course assumes that the current candidate subgraph could not be grown further while still ensuring that n instances could still be identified. If all the identical subgraphs with the target number of instances, n, are found, the compiler decreases the target number by one and starts the subgraph search again.

Additional conditions for the general subgraph search are that 1) the corresponding operations from each subgraph should be identical, 2) live values and immediate values should also be taken into consideration, and 3) inter-subgraph dependencies should not exist. Condition 1) enables the corresponding instructions inside the subgraphs to be packed into one opcode, and condition 2) enables packing whole operands of the instructions. Live values and immediate values are not generally considered in common subgraph pattern matching, but the SGLP compiler must take them into account because only same type of operands can be packed for SGLP. The last condition ensures that the subgraphs are parallelizable.

4.3 SIMD Lane Assignment

Once all possible groups of identical subgraphs are identified, the compiler selects the subgraphs to be packed and assigns them to SIMD lane groups. Instructions included in remaining subgraph groups lose the subgraph information and are reused in the next subgraph identification process. The objectives of SIMD lane assignment process are two-fold: 1) pack maximum number of instructions with minimum inter-lane data movement, and 2) ensure packed groups of instructions can be executed safely in parallel without any dependence violation. To achieve these goals, the compiler considers three kinds of criteria: gain, partial order, and affinity.

The gain of the subgraph is the most critical criteria and is largely calculated by the size of the subgraph. Larger subgraphs can provide higher performance with less overheads as more dataflow can be covered. The memory packing overhead is also accounted for in the gain if it incurs performance degradation. The compiler tries to assign subgraphs to specific SIMD lanes based on decreasing order of the gain.

The partial order between subgraphs inside the SIMD lane group is the next most critical issue. When assigning new identical subgraphs to different SIMD lane groups, the partial order of the subgraphs inside the SIMD lanes may be different across the SIMD lanes because identical subgraphs are only parallel with each other and the relations with other subgraph groups are not considered. If the relation between different subgraphs in some lane groups is different from the relations in other lane groups, the corresponding subgraphs cannot be executed in parallel. Figure 12 shows a simple example case of this kind of conflict. From a vectorized basic block having 3 groups of identical subgraphs with (A0, A1), (B0, B1), and (C0, C1), (A0, A1) and (B0, B1) are chosen to be parallelized using the two SIMD lane groups. After this assignment, C0 and C1 cannot execute in parallel through two SIMD lane groups because C0 must execute before B0 in the lane group 0-3 but C1 must execute after B1 in the lane group 4-7.

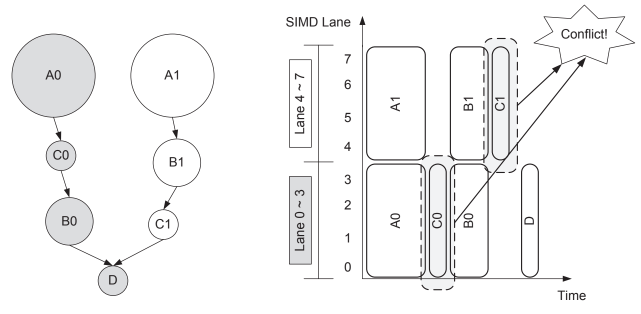


Figure 12. Subgraph partial order mismatch: when (B0, B1) is chosen to execute in different SIMD lanes, (C0, C1) cannot be chosen due to the partial order mismatch between lanes.

As the inter-lane data movement overheads inside the subgraphs are already solved by subgraph identification, the next objective is to minimize the overheads between different subgraphs. Typically, a subgraph is related to multiple other subgraphs, so the compiler must consider which combination of subgraphs can minimize the overall overhead. To address this issue, an *affinity cost* was introduced inspired by previous works [27, 28]. The affinity value for a pair of subgraphs reflects their proximity in the DFG. When a group of identical subgraphs is chosen to be parallelized, each lane group is assigned an affinity cost depending on how close the subgraph candidate is to any already placed subgraphs that have high affinity with the candidate. This gives preference for assigning a subgraph in the same lane group as other subgraphs it is likely to communicate with thus reducing inter-lane data movements.

$$\begin{aligned} affinity(A,B) &= \sum_{a \in nodes\_A} \Big(\sum_{d=1}^{max\_dist} 2^{max\_dist-d} \\ &\times ((N_{cons}(a,B,d) + N_{prods}(a,B,d)) \times C_0 \\ &+ (N_{com\_cons}(a,B,d) + N_{com\_prods}(a,B,d)) \times C_1)\Big) \\ &, where \quad C_0 >> C_1 \end{aligned} \tag{2}$$

Equation (2) calculates the affinity between two subgraphs A and B. The value is determined by four different relations between nodes inside A and B: producer, consumer, common consumer, and common producer relations. Producer/consumer relation means that nodes in A have direct/indirect producer-consumer/consumer-producer relations to nodes in B. Common producer/consumer relations mean that nodes in A and nodes in B have common producer/consumer relations. The former two relations have explicit data movement between subgraphs but the latter relations just imply that they may have some data movements when merging or diverging. Therefore, we put more weight on the former two relations ($C_0 >> C_1$). Nodes within max_{dist} are used, where N refers to the number of nodes in subgraph A that have a relationship with a node in subgraph B at a distance d. The distance is the number of nodes to reach the target node.

Algorithm 1 SIMD Lane Assignment

```
Input: IdSubGroups, G, SIMDGroups
Output: SIMDGroups
      { Assign subgraphs into the appropriate SIMD lane group.}
  1: SortSubGraphGroupsByGain(IdSubGroups);
  2: while HasGroup(IdSubGroups) do
  3:     curSubGroup ← Pop(IdSubGroups);
  4:     while HasGroup(curSubGroup) do
  5:         curSubGraph ← Pop(curSubGroup);
  6:         curSIMDGroup ←
             findSIMDGroupByMaxAffinity(SIMDGroups, curSubGraph);
  7:         curSIMDGroup → addSubGraph(curSubGraph);
  8:     end while
  9:     if (!PartialOrderCheck(SIMDGroups)) then
 10:         Restore(SIMDGroups);
 11:     end if
 12: end while

      { If no more updates, find the main lane group and assign remaining nodes.}
 13: if (!IsUpdated(SIMDGroups)) then
 14:     curSIMDGroup ←
         findSIMDGroupByMaxOverhead(SIMDGroups);
 15:     curSIMDGroup → addRemainingNodes(G);
 16:     SetMainSIMDGroup(curSIMDGroup);
 17: end if
```

Algorithm 1 shows how the SIMD lane assignment works. The inputs are the list of identical subgraph groups (*IdSubGroups*), dataflow graph (*G*) and the current list of SIMD lane groups (*SIMDGroups*). The output is the list of SIMD lane groups with new subgraph assignment (*SIMDGroups*). The algorithm starts by sorting the *IdSubGroups* by subgraph gain because we place the top priority on the gain of subgraph. Based on the sorted order of the list, the while loop assigns the subgraphs on the appropriate SIMD lane group. Lines 3-8 take one identical subgraph group and assign each of the subgraphs onto the SIMD lane group having the maximum affinity. Lines 9-11 perform the partial order check for all the SIMD lane groups and, if some conflicts occur, the latest update is cancelled. When no more subgraphs are assigned to the initial *SIMDGroups*, the compiler decides not to try the subgraph identification process again using the remaining nodes, sets the SIMD lane group with the maximum overhead as the main lane group, and assigns uncovered nodes of DFG to the main lane group in order to minimize the total overhead.

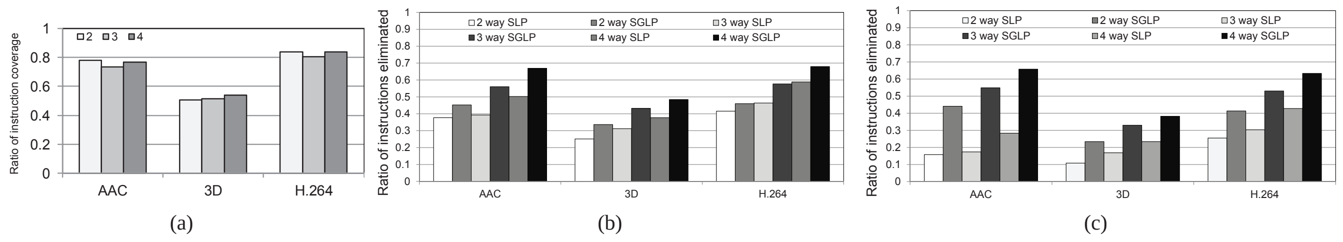


Figure 13. Ratio of instructions covered by the subgraph level parallelism and static instructions eliminated for three media applications: (a) instruction coverage, (b) static instruction elimination without inter-lane overheads, and (c) static instruction elimination with inter-lane overheads.

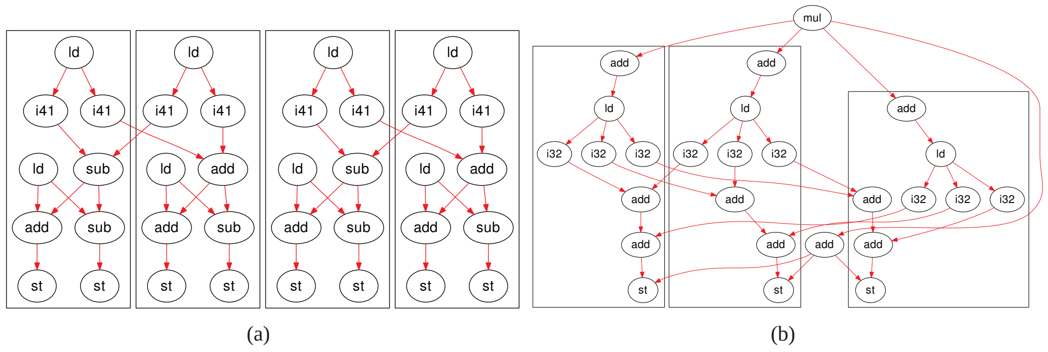


Figure 14. Example dataflow graphs: (a) FFT: two identical subgraphs ((1) ld, i41, i41, (2) ld, (sub/add), add, sub, st, st), (b) MatMul3x3: two identical subgraphs ((1) add, ld, i32 , i32, i32 (2) add, add, st). i41 and i32 are intrinsic instructions.

4.4 Code Generation

The compiler generates new vectorized IR from the lane assignment and inter-lane movement information from the previous process. When the compiler meets instructions covered by the identical subgraphs, the compiler gathers each parallel operand and converts them into one long register by remapping, a short immediate, or a wide constant. When a wide constant exists, the compiler generates the data and saves it to the constant memory. Shuffle instructions are also added if the compiler detects inter-lane data movement.

5. Experimental Results

5.1 Experimental Setup

To evaluate the availability and performance of SGLP, 144 loop kernels, varying in size from 4 to 142 operations, are extracted from three media applications in the embedded domain (AAC decoder, 3D graphics, and H.264 decoder). The iteration count per invocation of the kernels varies from 1 to 1024, and the natural SIMD widths are decided by the conditions discussed in Section 2.2.1 and memory dependence checks are performed using profile information. The IMPACT compiler [26] is used as the frontend compiler and both SGLP and SLP [17] are implemented in the backend compiler using a SODA-style [21] wide vector instruction set. The inter-lane move is performed using a single-cycle delay shuffle instruction, supporting data rearrangement in the SIMD RF as indicated by the permutation pattern similar to vperm (VMX) or vec_perm (AltiVec [30]). We also allow some similar instructions (e.g. add/sub) to be packed as common vector architectures allow this. The vectorizable kernels are automatically vectorized by loop unrolling and the evaluation is performed using the loop-level vectorized basic block. The wide SIMD architecture as discussed in Section 2.1 is used as the baseline architecture. The number of SIMD resources can vary from 16 to 64, while the number of memory banks are limited to four.

Our experiments do not apply SGLP more than 4-way. Two main reasons for this are: 1) the degree of ILP, the theoretical maximum gain of SGLP, is mostly smaller than four, and 2) only computation instructions can be SIMDized, and therefore the decreased ratio of computation to memory instructions causes the performance to be constrained by the AGU pipeline.

5.2 Subgraph Level Parallelism Coverage

We first calculate the percentage of instructions covered by identical subgraphs in order to gauge the availability of subgraph level parallelism. From the vectorized basic blocks of kernels, we found identical subgraphs ranging from 2-way to 4-way. The coverage is calculated as the number of instructions covered by the identical subgraphs. The results for three applications are shown in Figure 13(a). For H.264 and AAC, a large percentage of instructions is covered by identical subgraphs because high degrees of parallelism still exists even inside the vectorized basic block. Even though SGLP covers relatively small amount of instructions, more than 50% of instructions in the 3D application are still covered. Compared to other applications, the 3D application has a smaller degree of SIMD opportunity due to each instruction having a small number of parallel instructions with the same operation.

The interesting point here is that the coverage of the 3-way for AAC and H.264 applications is smaller than the 2- and 4-way. This is because most dataflow graphs have a tree structure and therefore 2 and 4 way are well matched but 3-way frequently misses some instructions when dataflow merges. For example, a dataflow graph of the FFT kernel is likely parallelizable in a 4-way, and thus 3-way exploration cannot find the profitable identical subgraphs in the one remaining flow as shown in Figure 14(a).

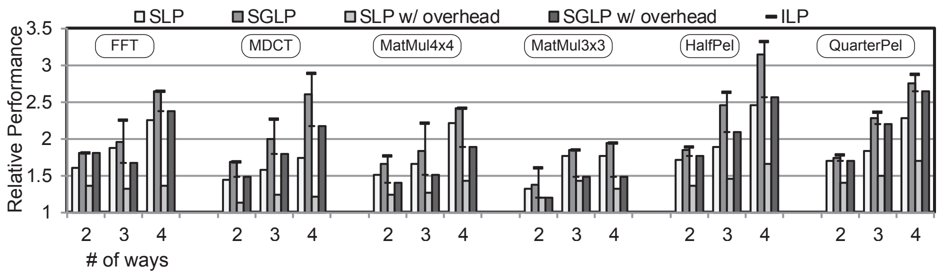


Figure 15. Performance comparison of SLP/SGLP without overhead, SLP/SGLP with overhead, and ILP for key kernels: FFT, MDCT for AAC, MatMul4x4, MatMul3x3 for 3D, and HalfPel, QuarterPel for H.264.

Figure 13(b) and Figure 13(c) show the ratio of static instructions eliminated from the vectorized basic block when applying SGLP and SLP. The configuration is expressed as: (number_of_simdization_ways) way (technique). Figure 13(b) shows the result without overhead (number of shuffle instructions) and the percentage of savings has the trend similar to that of the SGLP coverage. An interesting question is how the SGLP can eliminate more instructions than the SLP even though both techniques have a fair amount of gains. This is because 1) SLP frequently makes the wrong decision among various packing opportunities and 2) SLP cannot vectorize pure scalar codes [3]. When considering the inter-lane data movement overhead as shown in Figure 13(c), SLP performs dramatically worse than the ideal condition due to many shuffle instructions. On the other hand, SGLP was found to still deliver consistent amounts of instruction eliminations by smart data-movement control. Based on the application complexity, H.264 and 3D have a notable degradation of savings, whereas AAC is rarely affected by the overhead.

5.3 Performance

Inspired by the promising result of finding abundant opportunities for SGLP in the vectorized basic block, we compared the performance of SGLP to both SLP and ILP. Performances of SGLP and SLP are measured as the schedule length when the kernel is mapped a (a degree of loop-level vectorization $\times$ the number of ways)-wide SIMD architecture. As SGLP is the restricted form of ILP, the ILP result can be thought of as the theoretical upper bound. The performance of ILP is measured as the schedule length when the kernel is scheduled in the same sized fully-connected VLIW machine having a central register file. For example, if an example kernel is loop-level vectorized by 16 and 2-way SGLP is applied, the corresponding ILP performance is calculated when an ideal 32-wide VLIW machine executes unrolled scalar code.

Figure 15 and 16 show the individual performance enhancement results of six well-known kernels and geometric mean of gains for each application. The target ways are shown on the X-axis, relative performance normalized to the original vectorized kernel on the Y-axis. The following techniques are examined and shown as a bar form: SLP and SGLP with zero-cycle data-movement latency (SLP and SGLP) and SLP and SGLP with single-cycle data-movement latency (SLP and SGLP w/ overhead). The ILP results are also shown as a short horizontal form and the vertical line indicates the performance difference between ideal ILP and loop-level vectorization combined with practical SGLP. From these two graphs, substantial amounts of speedups exist in both ideal cases and are similarly scalable as ILP. In addition to this, gains from SGLP in real situations are also mostly prominent and scalable without large overhead increases on wider ways. In contrast, SLP with overhead has a large performance degradation due to the immense inter-lane data-movements, and increasing overheads on wider ways make it barely scalable.

Unlike most cases, a few kernels showed negligible performance improvements while applying SGLP, namely FFT in AAC and 3x3 matrix multiplication in 3D. These are due to the specific characteristics of each dataflow graph. First, as shown in Figure 14(a), the FFT kernel can have two subgraphs without inter-lane data movements in the 2-way case. In the 4-way case, each subgraph for the 2-way case is split once more with only two data movements such as (i41 $\rightarrow$ add) and (i41 $\rightarrow$ sub). In the 3-way case, three of the subgraphs for the 4-way case are identified and a remaining subgraph cannot be effectively executed in multi-lane, which has a high data-movement overhead. As a result, the gain of 3-way SGLP is worse than that of 2-way SGLP including overheads. Second, the 3x3 matrix multiplication can be split into three subgraphs as shown in Figure 14(b), and therefore a considerable increasing in overheads when applying 4-way SGLP hinder it from fully exploiting the benefits.

As shown in Figure 16, on average, SGLP without and with overheads achieve relative performance improvements of 1.42x, 1.36x at 2-way, 1.61x, 1.47x at 3-way, and 1.84x, 1.66x at 4-way. In addition to this, SGLP with overheads also provides 18-40% more performance improvement over baseline compared to SLP with the same resources. The performance difference between SGLP and SLP increases as applied on wider ways. Finally, a comparison with ILP suggests that SGLP is a cheap and powerful solution to accelerate performance, considering that SGLP only requires minimum additional hardware while a wide fully-connected VLIW architecture is impractical.

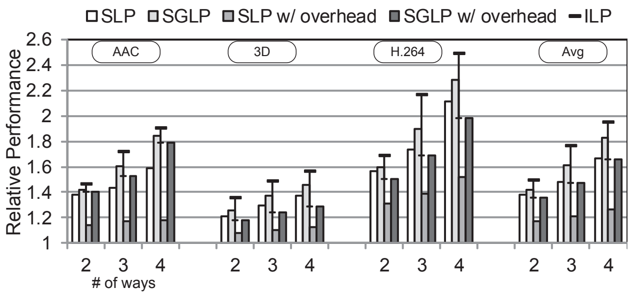


Figure 16. Average kernel performance comparison of SLP/SGLP without overhead, SLP/SGLP with overhead, and ILP for three application domains.

Based on the schedule results for kernels, we execute three applications on three wide SIMD architectures having 16, 32, and 64 lanes. When the original SIMD width of the kernel is equal or larger than the width of the architecture, SGLP or SLP is not exploited. Only when the current SIMD width of the kernel is insufficient to fully use the architecture, the available amount of SGLP, up to 4-way, is exploited. For example, 4-way SGLP is exploited if a kernel is 4-wide vectorized on the 16 way architecture and 2-way when 8-wide vectorized. The final performance is also compared to traditional ILP in the equivalent VLIW architecture and SLP. The results are provided in Figure 17 with considerable performance gains. The X-axis shows the number of SIMD lanes on the wide SIMD architecture and the Y-axis shows speedup relative to the simple SIMD execution time on the baseline architecture. The two bars of each application represent the runtime speedup of real SLP and SGLP with overheads. In a similar ways from previous Figures, ILP results are also provided. For all the applications, real SGLP still shows notable performance gain by utilizing more SIMD resources with smart overhead control. As we discussed in Section 2.2.2, kernels having smaller than 16 SIMD width are accelerated by SGLP when using 16 wide architecture, and AAC and H.264 have high gains due to the high execution time ratio of such kernels, which are more than 50% of their total execution time. As the architecture size becomes larger, the performance is saturated at some point because SGLP is constrained by maximum degree of 4. The key observation is that the real performance gain of SGLP is also fairly scalable due to the fact that the performance gain successfully compensates for the increased overheads different from SLP. Finally, on average, SGLP with overhead can have 1.61x, 1.73x, and 1.76x speedups at 16, 32, and 64 wide SIMD architectures while SLP only achieves 1.24x, 1.28x, and 1.29x speedups.

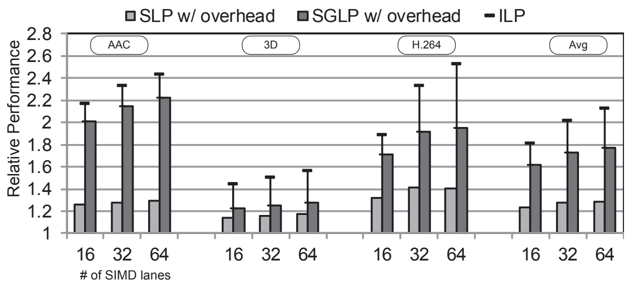


Figure 17. Overall performance comparison of SLP/SGLP with overhead and ILP for three domains on SIMD architectures.

5.4 Energy Measurement

To evaluate the energy savings of SGLP in the real world, we measured total energy consumption for running H.264 to determine the effectiveness of SGLP. We used a 32-wide SIMD architecture for SGLP, and a practical 4-way VLIW, in which each datapath supports 8-wide SIMD instructions for ILP and an 8 read-ports, 4 write-port 8-wide SIMD RF. Both architectures are generated in RTL Verilog for a 200 MHz target frequency, then synthesized with the Synopsys Design Compiler and Physical Compiler using IBM 65nm standard cell library with typical operating conditions. PrimeTime PX is used to measure power consumption. Instead of measuring power for every cycle, average activity of each component was monitored. Figure 18 shows that the SGLP is 30% more energy efficient than ILP. Even though the performance of SGLP is slightly lower, the high power overheads of VLIW implementation, such as those introduced by a multi-port register file and a complex interconnect, dominate the results. The power number for constant memory is also considered based on the standby power and read power extracted from the SRAM compiler. The constant memory power overhead is trivial because the standby power is roughly 1/250 of read power and the wide constant values are rarely read. The access timing is also smaller than 5 ns (i.e., 200 MHz), hence data can be read in one cycle.

	SGLP @ 32-wide SIMD	ILP @ 4 way 8-wide VLIW	ratio
power (mW)	54.40	93.17	58.39%
cycle(million)	13.07	10.77	121.36%
energy (mJ)	3.55	5.02	70.86%

Figure 18. Energy comparison for the SGLP on the 32-wide SIMD architecture and ILP on the 4 way 8-wide VLIW architecture.

6. Related Works

Most prior work in automatic vectorization are performed on the loop-level [2, 25] and some of the techniques have already been implemented on commercial compilers such as the Intel Compiler [15]. These types of vectorization are usually exploited by loop unrolling. Our SGLP vectorization starts after simple loop-level vectorization, and thus it is an orthogonal approach and can be helpful to enhance the overall performance of our compiler framework by finding more loop-level DLP. SGLP tries to identify opportunities for parallelism within the vectorized basic block.

Superword-level parallelism [17] is the closest related work but this work is hard to apply to long vector architectures as discussed in Section 3. To improve this technique, some research [18, 31] focuses on smart memory control such as increasing contiguous memory instructions and decreasing memory accesses. There are two key differences between SGLP and SLP: 1) SGLP tries to minimize the SIMD overheads in the scope of dataflow graph analysis, whereas most approaches do in the scope of memory management, and 2) we focus on finding groups of instructions to guarantee sufficient gain over the overheads but others usually focus on decreasing the overheads. Unroll-and-jam with SLP [25] is the most similar work and we can get 30% higher performance on average due to SLP being less effective when applied to scalar code.

Another key contribution of this work is the ability to minimize the interaction between the SIMD lanes. This scheme is highly related to the research in the area of clustering [1, 6, 7, 13]. However, general clustering techniques for VLIW machines focus on load balancing and critical path search, and thus cannot handle dataflow and instruction mismatches between clusters.

Subgraph exploration for finding identical subgraphs is also a well-known research area [8–11] but the goal of these works is mostly to generate custom accelerators for the subgraphs. We introduce a new algorithm for orchestrating sets of subgraphs at a high-level for SIMDization on existing architectures.

AnySp [34] or SCALE [16], which exploit multiple forms of parallelism, are also similar to this work. AnySp integrated DLP and TLP, and SCALE exploits both vector parallelism and TLP. However, they need substantial architectural changes like multiple AGUs, flexible functional units, and swizzle networks in AnySp, or additional multiple fetch unit, special inter-cluster network, and Atomic Instruction Block (AIB) cache in SCALE. However, we can provide SGLP with two minimal hardware modifications (a small wide-constant memory and banked memory access) that incur very little overhead.

7. Conclusion

The popularity of mobile computing platforms has led to the development of feature-packed devices that support a wide range of software applications with high single-thread performance and power efficiency requirements. To efficiently achieve both objectives, embedding SIMD components is an attractive solution, However, utilization of SIMD resources is a major limiting factor for adopting such a scheme. In response, we propose an efficient vectorization framework, called the *SIMD defragmenter*, to enhance the throughput by maximizing SIMD utilization. The *SIMD defragmenter* framework first performs simple loop-level vectorization, then tries to find more DLP within the vectorized basic block using subgraph level parallelism (SGLP). To achieve this, partially parallelizable subgraphs are identified inside the basic block, which are offloaded to the unused SIMD lanes while minimizing the number of inter-lane data movements. We introduce a new way to orchestrate the partially parallel subgraphs, which is implemented in our SGLP compiler. The SGLP compiler is able to effectively assign the SIMD lanes for each subgraph based on the relations between subgraphs. On a 16-wide SIMD processor, SGLP obtains an average 62% speedup over traditional vectorization techniques, with a maximum gain of 2x. In comparison to superword-level parallelism, the well-known basic block level vectorization technique, SGLP achieves an average 30% speedup. We believe as SIMD, or more general data-parallel, accelerators become more commonplace, new techniques to put these resources to work across a wide spectrum of applications will be essential.

8. Acknowledgments

Thanks to Anoushe Jamshidi, Mark Woh, Shuguang Feng, Griffin Wright, and Gaurav Chadha for all their help and feedback. We also thank the anonymous referees who provided good suggestions for improving the quality of this work. This research is supported by Samsung Advanced Institute of Technology and the National Science Foundation under grants CCF-0819882 and CNS-0964478.

References

[1] A. Aletà, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Dec. 2001.

[2] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[3] R. Barik, J. Zhao, and V. Sarkar. Efficient Selection of Vector Instructions Using Dynamic Programming. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, Dec. 2010.

[4] K. Berkel, F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal Applied Signal Processing*, 2005(1):2613–2625, 2005.

[5] H. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. In *Intl. Symposium on System-on-a-Chip*, pages 15–20, Nov. 2003.

[6] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1992.

[7] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[8] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[9] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[10] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 147–157, Oct. 2006.

[11] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2003.

[12] J. Glossner, E. Hokenek, and M. Moudgill. The sandbridge sandblaster communications processor. In *Proc. of the 2004 Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[13] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–23, Oct. 2000.

[14] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proc. of the 2004 International Symposium on Low Power Electronics and Design*, pages 32–37, Aug. 2004.

[15] Intel. Intel compiler, 2009. software.intel.com/en-us/intel-compilers/.

[16] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, 2004.

[17] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[18] S. Larsen and S. Amarasinghe. Increasing and detecting memory address congruence. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, Sept. 2002.

[19] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[20] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.

[21] Y. Lin et al. Soda: A high-performance dsp architecture for software-defined radio. *IEEE Micro*, 27(1):114–123, Jan. 2007.

[22] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin. Dynamic power gating with quality guarantees. In *Proc. of the 2009 International Symposium on Low Power Electronics and Design*, pages 377–382, Aug. 2009.

[23] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram. A case for guarded power gating for multi-core processors. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, Feb. 2011.

[24] D. Nuzman et al. Vapor simd: Auto-vectorize once, run everywhere. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, pages 151–160, Apr. 2011.

[25] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short simd architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–11, 2008.

[26] OpenIMPACT. The OpenIMPACT IA-64 compiler, 2005. http://gelato.uiuc.edu/.

[27] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–146, Oct. 2006.

[28] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008.

[29] J. Park, D. Shin, N. Chang, and M. Pedram. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In *Proc. of the 2010 International Symposium on Low Power Electronics and Design*, pages 419–424, Aug. 2010.

[30] F. Semiconductor. Altivec, 2009. www.freescale.com/altivec.

[31] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 45–55, 2005.

[32] D. Talla, L. K. John, and D. Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, 2003.

[33] M. Woh et al. From SODA to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008.

[34] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 128–139, June 2009.